Modelling of Software intensive systems

Assignment 4 Statecharts

First Edition

L. Jason	Jason.Liu@student.uantwerpen.be	s0213082
S. Kin Ning	KinNing.Shum@student.uantwerpen.be	s0202054

2024 - 12 - 21

Contents

1 | Exercises 3

1.1 Exercise A 3

1.2 Exercise B 4

1.3 Exercise C 5

1.4 Exercise D 6

1.5 Exercise E 7

2 | Assignment 8

2.1 Chart 8

2.2 Scenarios 9

2.3 Workflow 10

1 | Exercises

1.1 | Exercise A

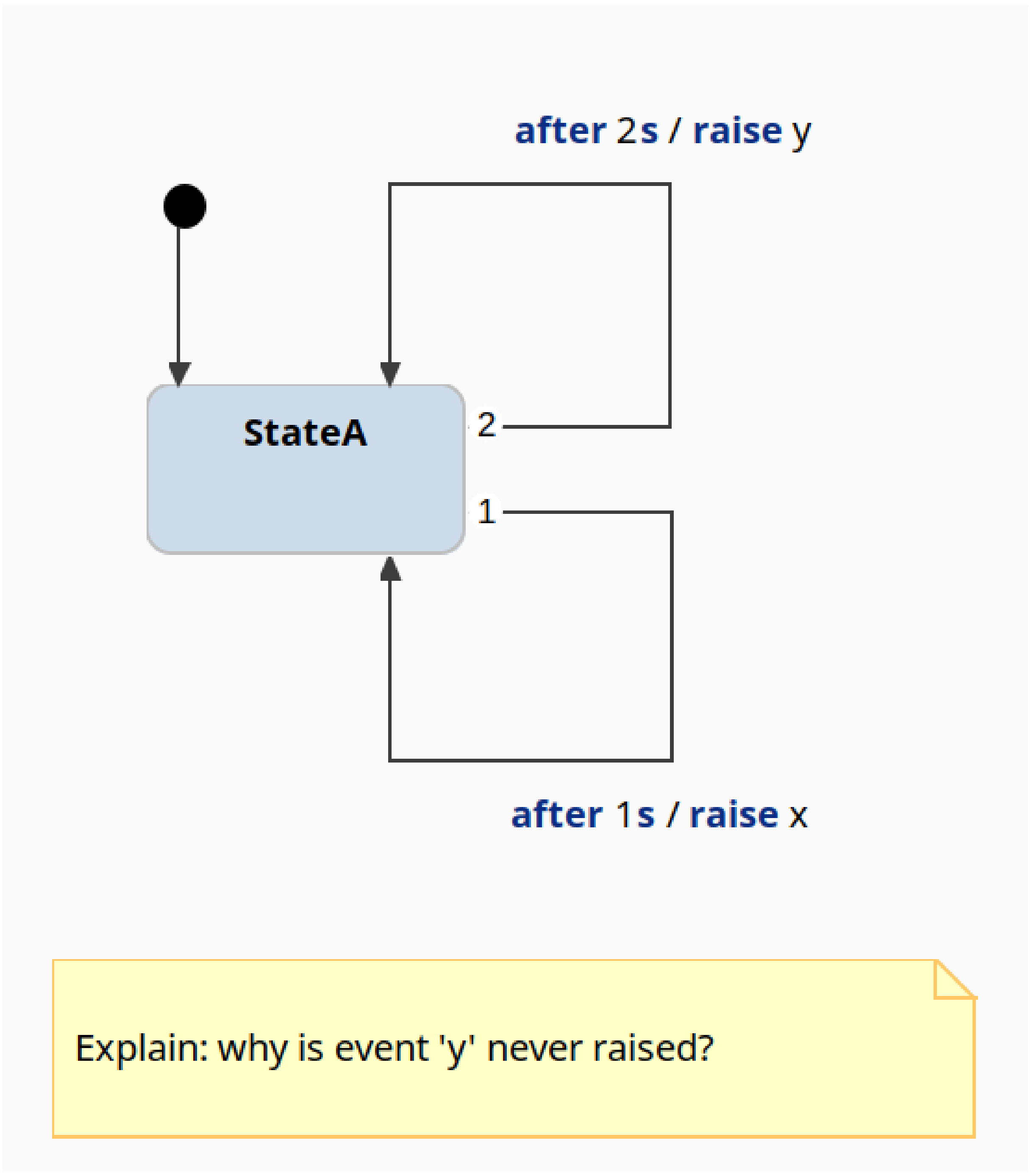
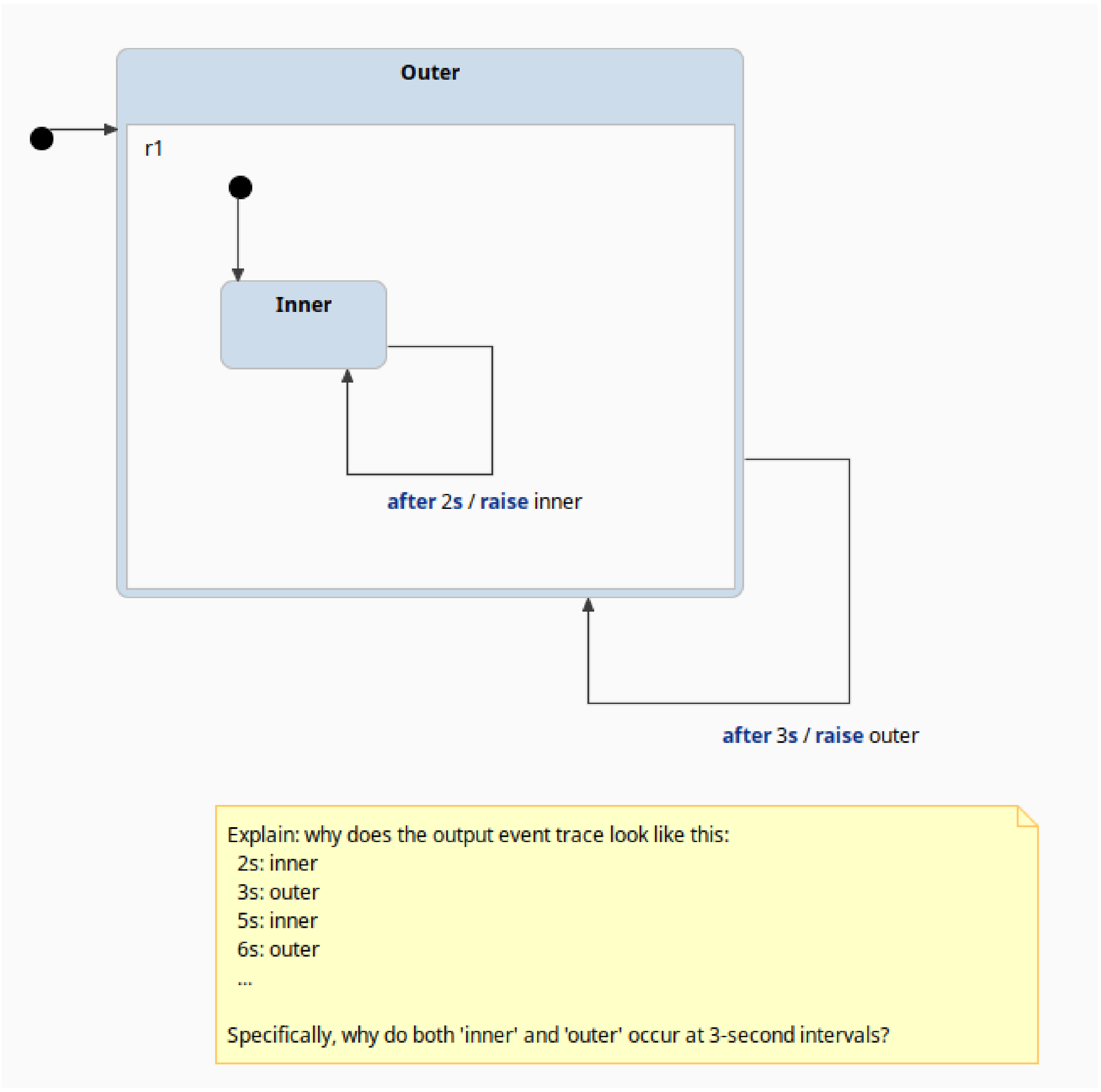


Figure 1: Exercise A statechart

‘y’ is never raised because ‘x’ gets raised first. After ‘x’ is raised, the timer is reset for both events.

1.2 | Exercise B



Explain: why does the output event trace look like this:
2s: inner
3s: outer
5s: inner
6s: outer
...

Specifically, why do both 'inner' and 'outer' occur at 3-second intervals?

Figure 2: Exercise B statechart

This is because the Outer state is the parent state of the Inner. While both happen in-synchronically in theory, when entering the inner state the timer gets reset. So first the inner state occurs because this timer is shorter than the outer state. The outer state occurs next, this causes the inner timer to reset, so originally it was 1 but now it is back to zero. Now after 2 seconds the inner state occurs again and the cycle repeats.

The alignment after every 3 seconds occur because the outer state keeps resetting the inner state timer.

1.3 | Exercise C

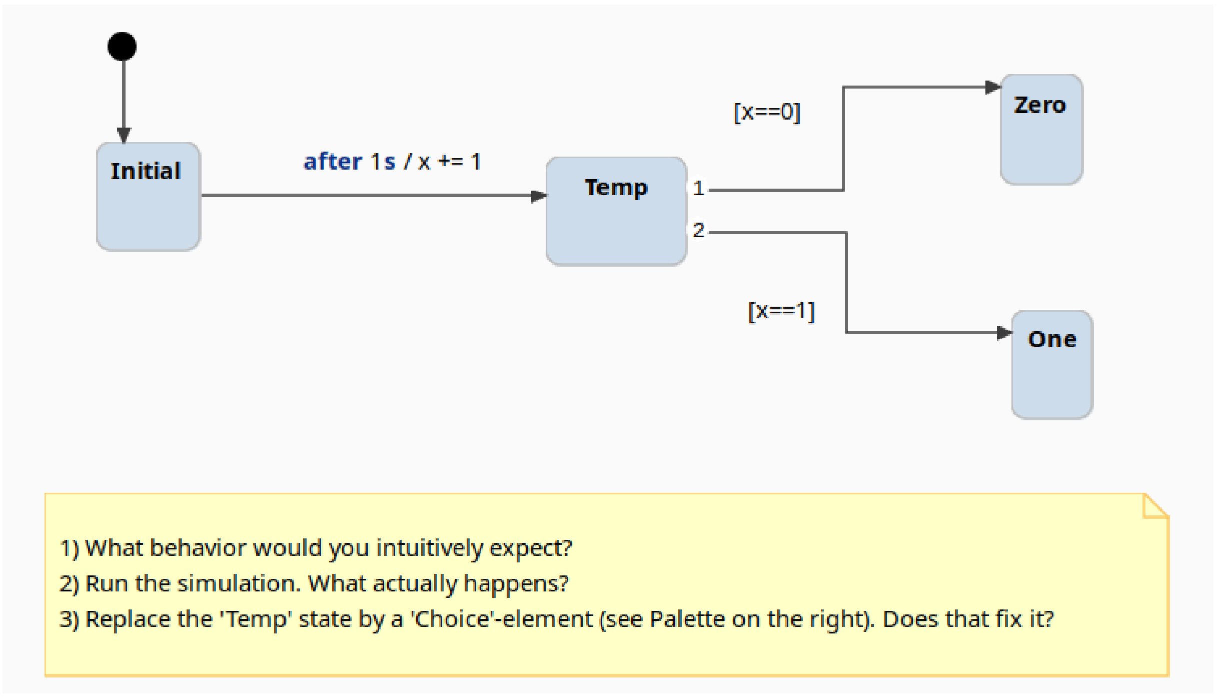

Initial
after 1s / x += 1
Temp
1
2
[x==0]
Zero
[x==1]
One
1) What behavior would you intuitively expect?
2) Run the simulation. What actually happens?
3) Replace the 'Temp' state by a 'Choice'-element (see Palette on the right). Does that fix it?

Figure 3: Exercise C statechart

1. We know that it starts from the initial state, and assuming that the value is zero in the beginning, we add it by one and afterwards temp goes to One. It never reaches state zero because from the initial state we never were zero.
2. The statechart does not directly transition from temp to either state. It stays in Temp indefinitely.

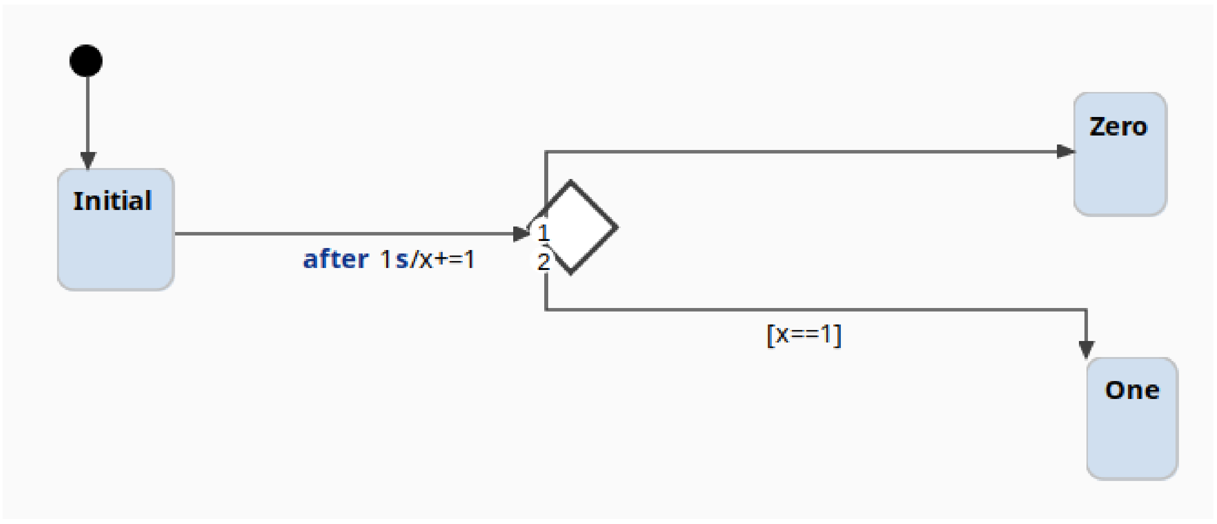


Figure 4: Solution to exercise C

3. Yes, it does indeed fix it. Looking at figure 4 we can see that now based on the value it makes a choice. The choice element makes a decision based on the changes that happen on variable x. Note that we did not add an extra guard to the zero transition, this is because the choice element prefers having a default transition. Works the same if the guard was added.

1.4 | Exercise D

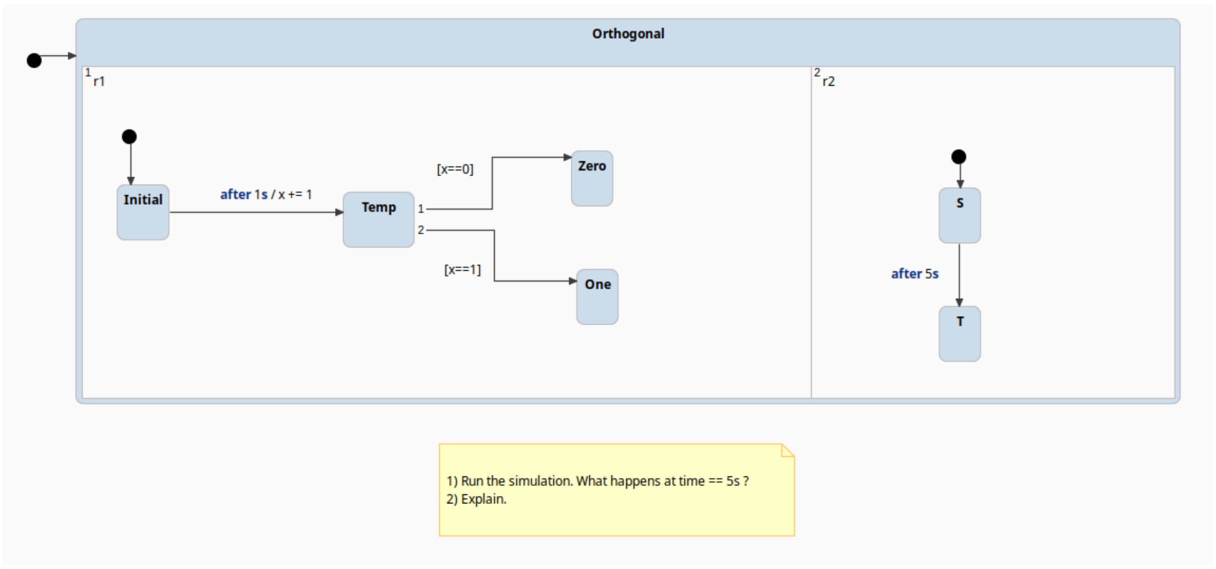
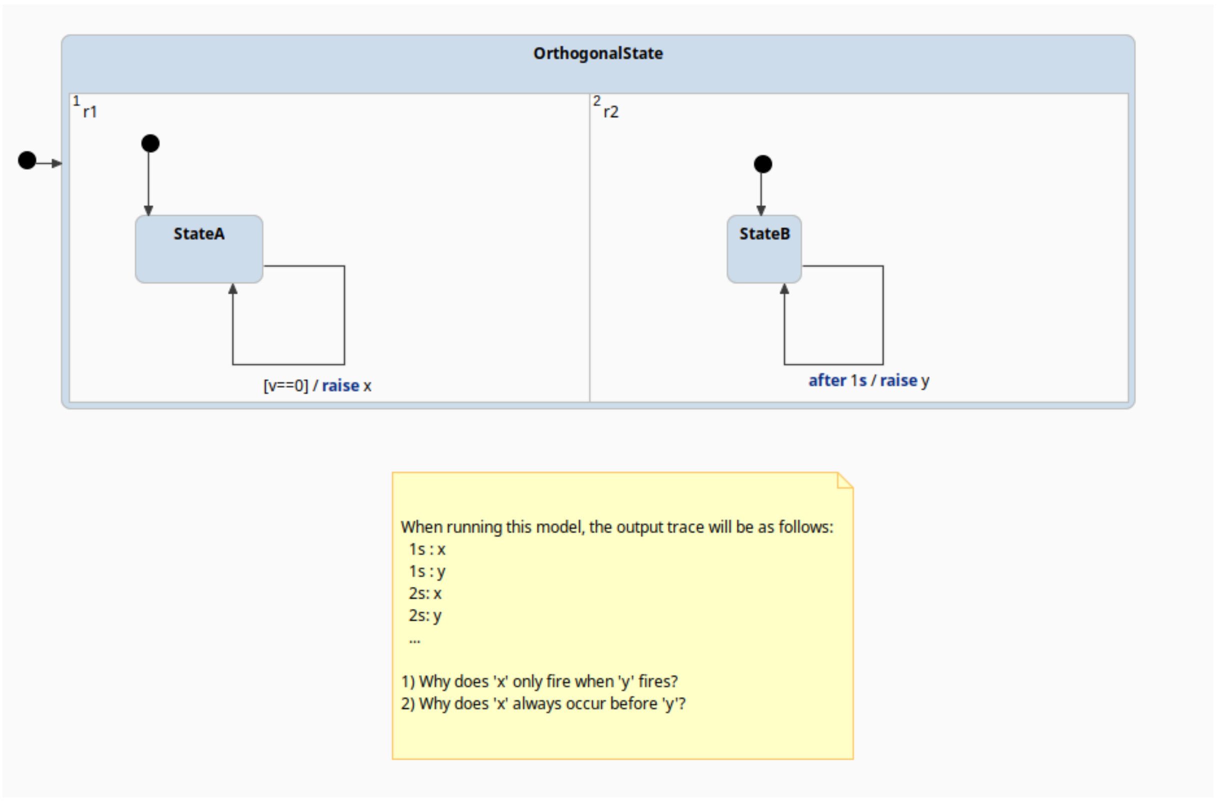


Figure 5: Exercise D statechart

1. After one second, we reach Temp with value 1 on x. Like before it stays stuck here until ‘r2’ updates. After 4 more seconds ‘r2’ updates and so does ‘r1’. In ‘r1’ we move from Temp to One.
2. Why does this happen? This is because after ‘r2’ updates the whole orthogonal state gets reevaluated. This causes any other state in the orthogonal state to be reevaluated to, so any state that is guarded by some transition will check if it can move. This causes in our case that ‘r1’ moves to One.

1.5 | Exercise E



When running this model, the output trace will be as follows:

```
  1s : x
  1s : y
  2s: x
  2s: y
  ...
```

1) Why does 'x' only fire when 'y' fires?
2) Why does 'x' always occur before 'y'?

Figure 6: Exercise E statechart

1. The reasoning is almost the same as the one behavior. Because the raise is specified based on a guard, we evaluate after every ‘trigger’. The second region triggers after every 1 second, but this makes the whole orthogonal state be reevaluated. So region one also gets reevaluated and that is why it gets triggered at the same time.
2. This is because of the order of what gets evaluated first. The priority set in Itemis Create is based on left to right and top to bottom ordering. So even if region 2 is the trigger, the evaluation happens based on the order of the orthogonal state.

2 | Assignment

2.1 | Chart

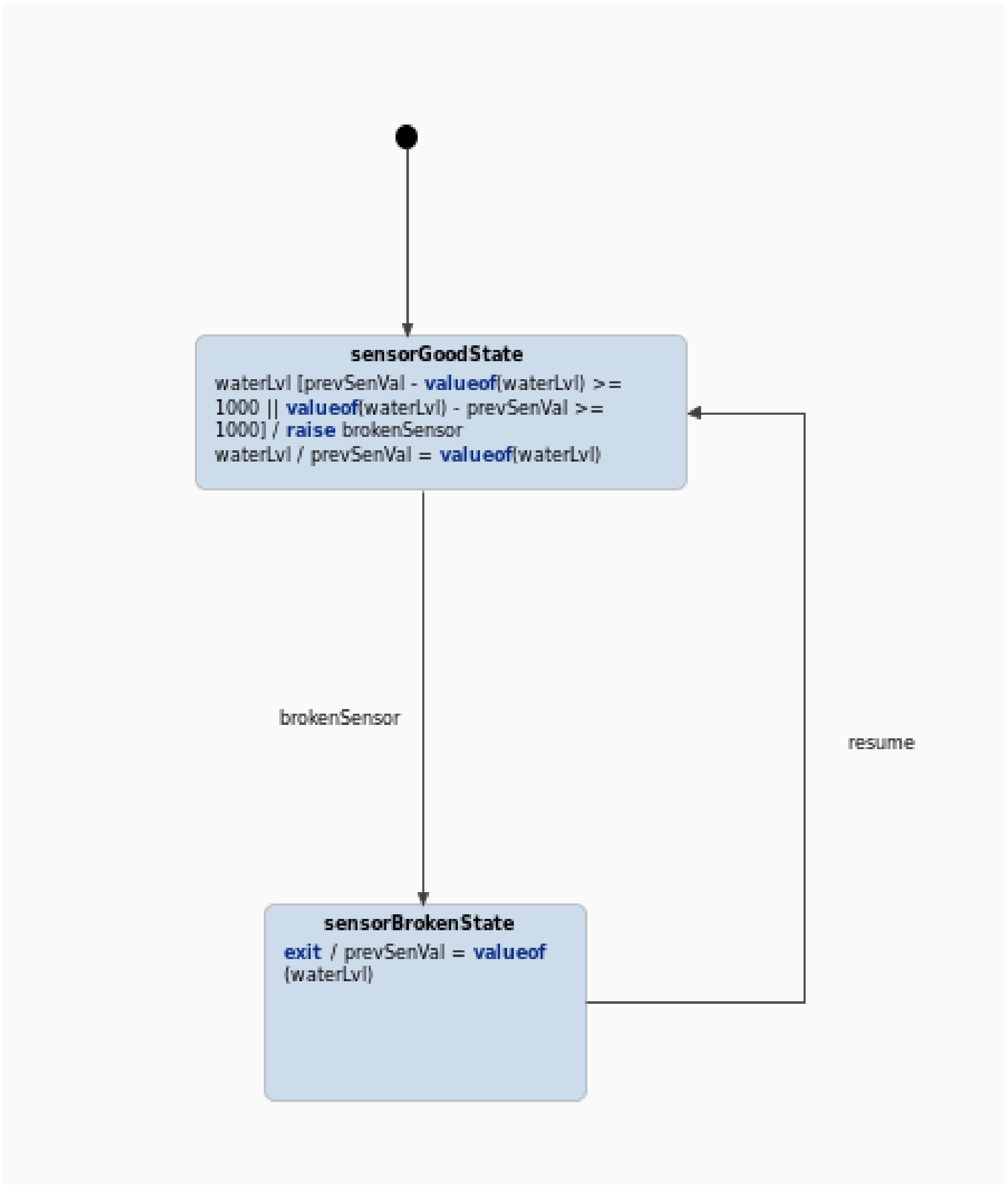

sensorGoodState
waterLvl [prevSenVal - valueof(waterLvl) >= 1000 || valueof(waterLvl) - prevSenVal >= 1000] / raise brokenSensor
waterLvl / prevSenVal = valueof(waterLvl)
brokenSensor
resume
sensorBrokenState
exit / prevSenVal = valueof (waterLvl)

Figure 7: Chart that detects sensor defects.

Every time the sensor reads an input, it compares it to the previous value and raises `brokenSensor` if the difference is greater than 1000.

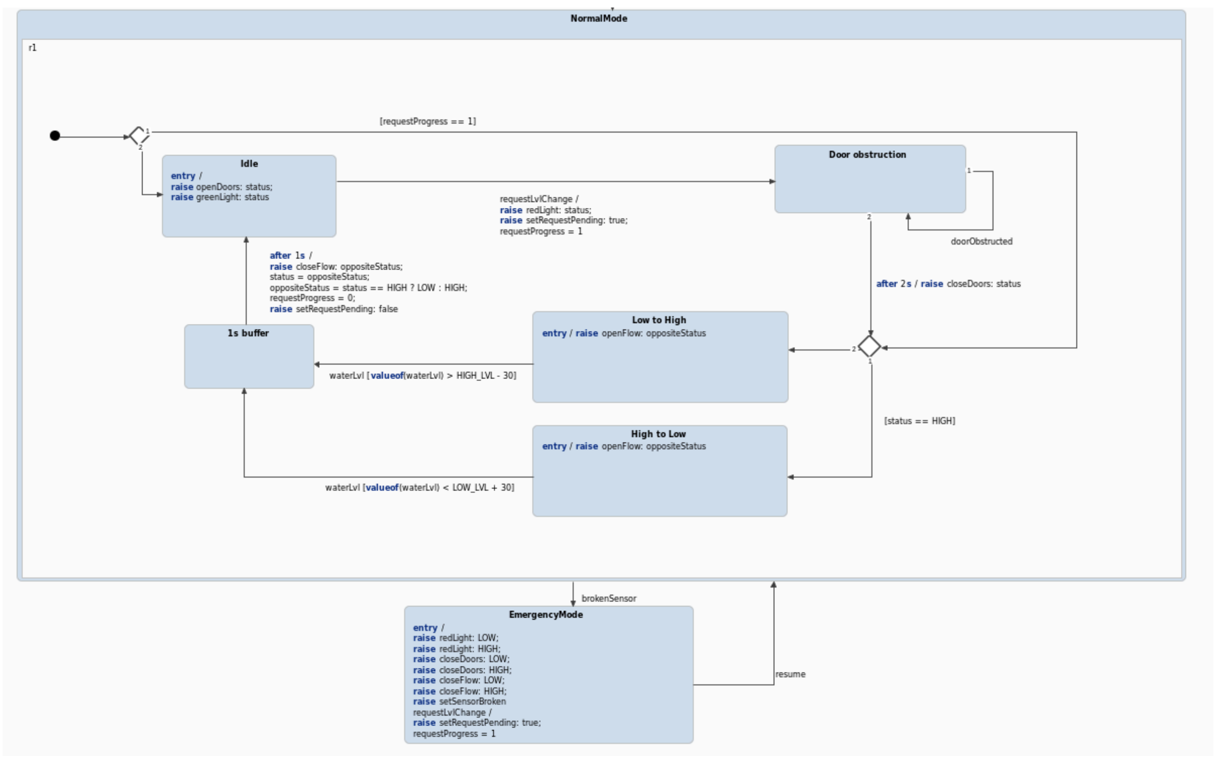


Figure 8: Chart that simulates the lock.

Both charts are in different regions of an orthogonal state, so they run in tandem. This chart starts at idle and changes state after a change request. The doors close after 2 seconds, unless the doors are obstructed, then it will close 2 seconds later. After closing, it will either enter the state `Low to High` or `High to Low`, depending on the water level. We keep track of a status to see whether it should go High or Low. After the required water level is reached, it will wait 1 second and return to idle.

If at any point the sensor defects, it will enter emergency mode and returns to normal operations after resuming. It keeps track if a request has been made or is in progress. If it is, it will skip to `Low to High` or `High to Low`.

2.2 | Scenarios

All tests passed, and we added 2 scenarios to test requirements.

2.2.1 | Scenario 1

This scenario tests the door obstruction, followed by a sensor defect.

```
# Test obstruct door
"name": "obstruct door and break sensor while obstructing",
"input_events": [
    (0, "water_lvl", 500),
    (1000000000, "request_lvl_change", None),
    (2500000000, "door_obstructed", None),
    (4000000000, "water_lvl", 10000),
    (4500000000, "water_lvl", 1000),
    (5000000000, "resume", None),
    (6000000000, "water_lvl", 1500)
],
"output_events": [
    (0, "open_doors", 0),
    (0, "green_light", 0),
    (1000000000, "red_light", 0),
    (1000000000, "set_request_pending", True),
    (4000000000, "close_doors", 0), # close doors not after 2s, but because it enters
emergency mode
    (4000000000, "set_sensor_broken", None),
    (5000000000, "open_flow", 1), # open correct flow, door stays closed
    (7000000000, "close_flow", 1),
    (7000000000, "set_request_pending", False),
    (7000000000, "open_doors", 1),
    (7000000000, "green_light", 1),
],
```

As there were no tests that checks if the door obstruction works, this test covers it. It checks if the doors close 2 seconds after a level change request. It also tests if it will enter emergency mode in ‘door closing mode’.

2.2.2 | Scenario 2

This scenario tests if it accepts requests in emergency mode.

```
# Test if it can accept requests when in emergency mode and proceed with the request
when it resumes.
"name": "Break sensor, request water change, then resume",
"input_events": [
    (0, "water_lvl", 500),
    (500000000, "water_lvl", 10000),
    (1000000000, "request_lvl_change", None),
    (1500000000, "water_lvl", 1000),
    (2000000000, "resume", None),
    (3000000000, "water_lvl", 1500)
],
"output_events": [
    (0, "open_doors", 0),
    (0, "green_light", 0),
    (500000000, "red_light", 0),
    (500000000, "close_doors", 0),
    (500000000, "set_sensor_broken", None),
    (1000000000, "set_request_pending", True), # accept request
    (2000000000, "open_flow", 1), # immediately open the correct flow, doors stay
closed
    (4000000000, "close_flow", 1),
    (4000000000, "set_request_pending", False),
    (4000000000, "open_doors", 1),
    (4000000000, "green_light", 1),
]
```

First, it breaks the sensor, then it makes a water level change request. This will test if a request can be accepted after entering emergency mode. After resuming, check if the correct flow opens.

Both scenarios pass in our implementation.

2.3 | Workflow

We split the assignment in 2 parts and each person made 1 part.

- Jason solved all exercises and made the initial design of the state chart. Time spent: 5
- Kin Ning finished and fixed the design, and did the tests and scenarios. Time spent: 6